DRAGON BALL. Episodio 3

Después de que Jacu investigara el servidor de la empresa Capsule Corp con la ayuda del Dr.Brief, las sospechas de Trunks quedaron confirmadas. La existencia de un fichero sospechoso que no ha sido creado por nadie de la empresa, parece indicar que ésta ha sido comprometida. Para aclarar el enigma, el Dr.Brief pide a Bulma que estudie el fichero en cuestión y extraiga cualquier información relevante que pudiera arrojar luz sobre el caso. Bulma consigue obtener la información de su creador, el Dr. Raichi, y descubre, además, que el contenido del fichero está escrito en una extraña lengua de la raza Tsufur. Para descifrar el contenido y obtener el texto en un lenguaje que ellos comprendan necesitan una clave. ¿Podrás encontrarla?

Descarga del fichero:
https://drive.google.com/file/d/1Uihvl5nEjkarfM03DV8J5RgJ2ZD1zy_T/view?usp=sharing

Info: La flag tiene el formato UAM{md5 del string encontrado}

Resolución

Descargamos el fichero ***main.***

Analizamos

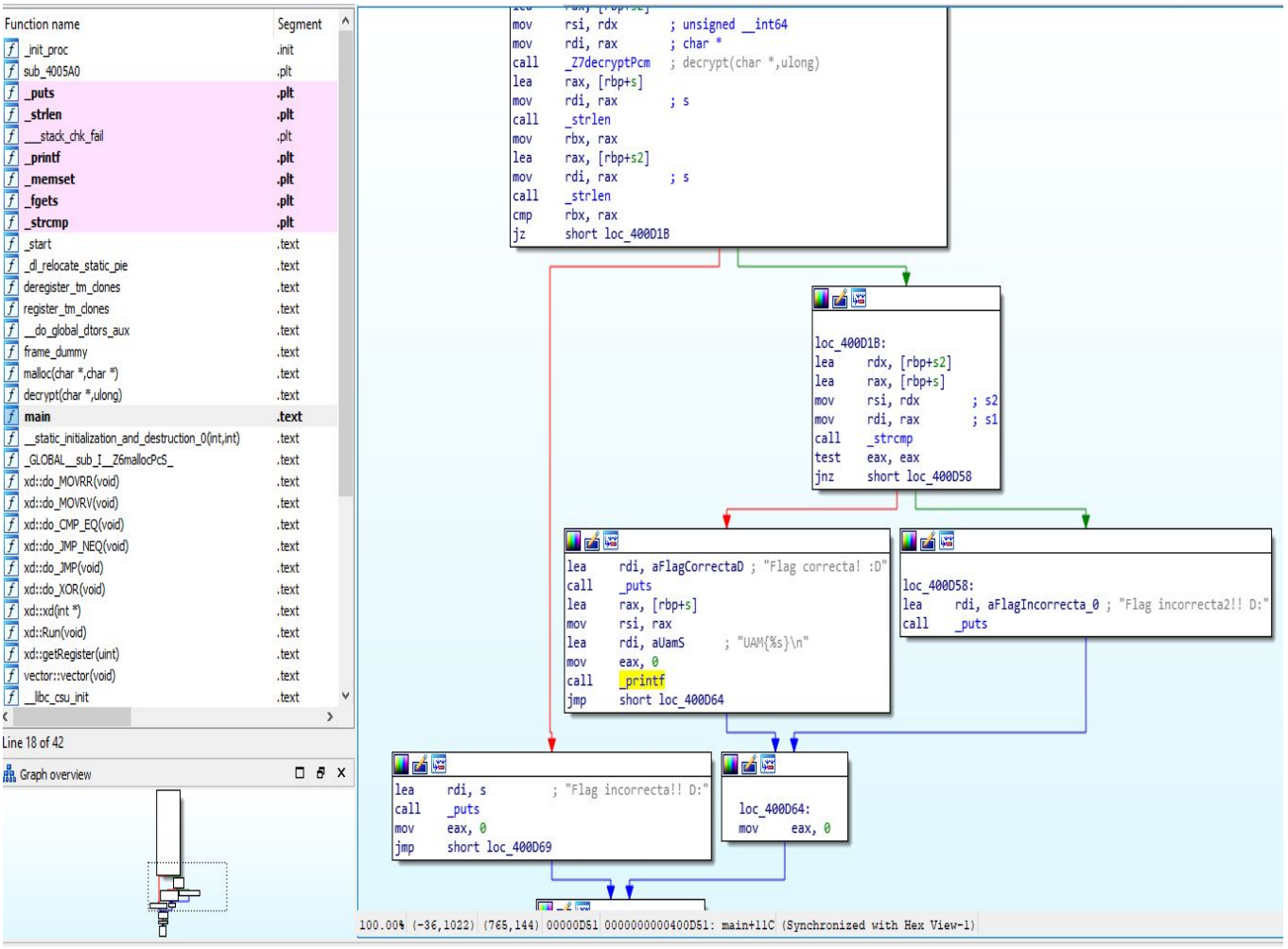
file main

main : ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0, BuildID[sha1]=6c62b346db422600ce26c67127d49dbb77ba4878, not stripped

Probamos funcionamiento:
./main
Flag: 00000000
Flag incorrecta!! D:

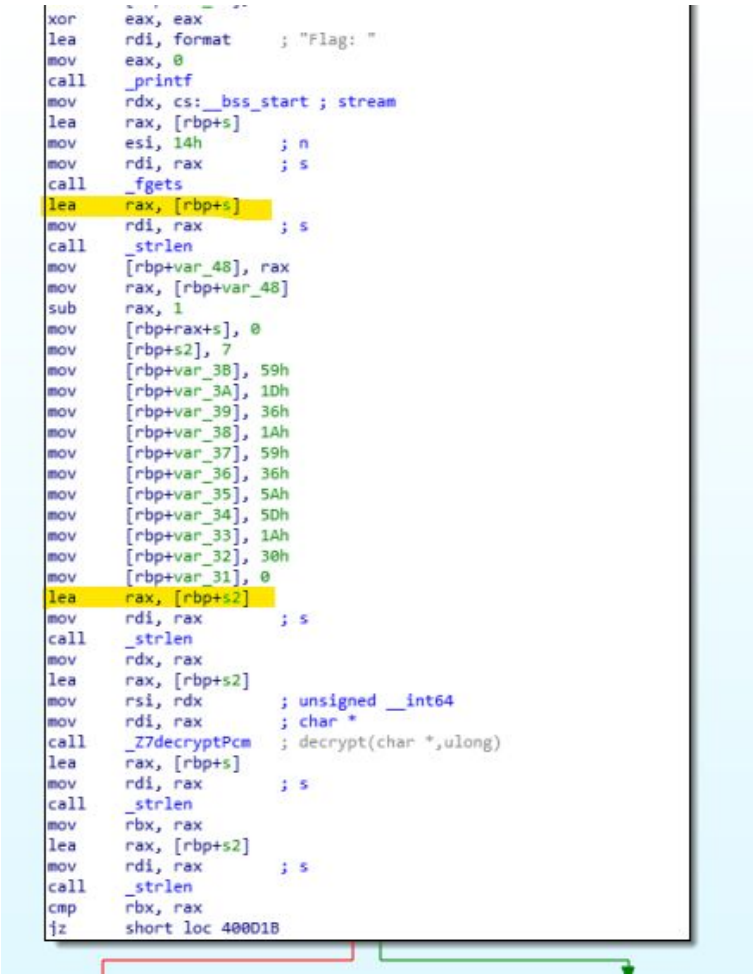
Pasamos al análisis estático con la versión gratuita de Ida.

Empezamos por la función main, donde aparece la cadena (“flag incorrecta!! ”)

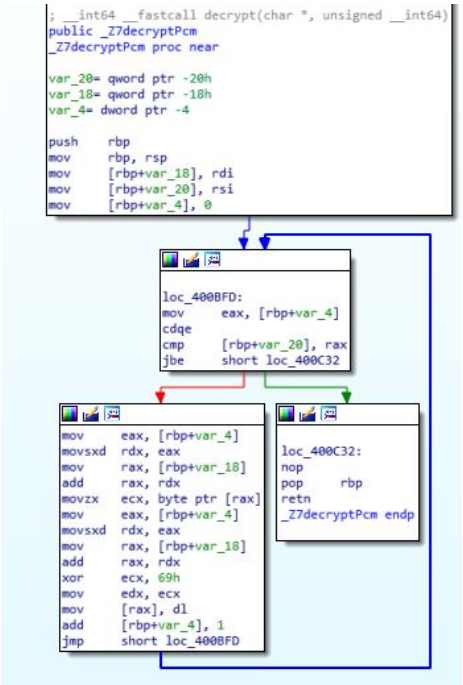


La función main, nos muestra el flujo del programa, en el que observamos que para llegar a la flag correcta se hacen dos comprobaciones, una primera con el tamaño de la cadena _strlen y una segunda comparando dos cadenas _strcmp.

Estudiemos las cadenas comparadas s, s2 (rbp+s2 y rbp+s).



La primera cadena, [rbp+s] es la que introducimos nosotros _fgets.
La otra, se establece inicialmente en el código [7,59,1d,36,1a,59,36,5a,5d,1a,30], después se modifica con la función (**Z7decrytpPcm**), donde pasamos como parámetros, la cadena y su tamaño _strlen



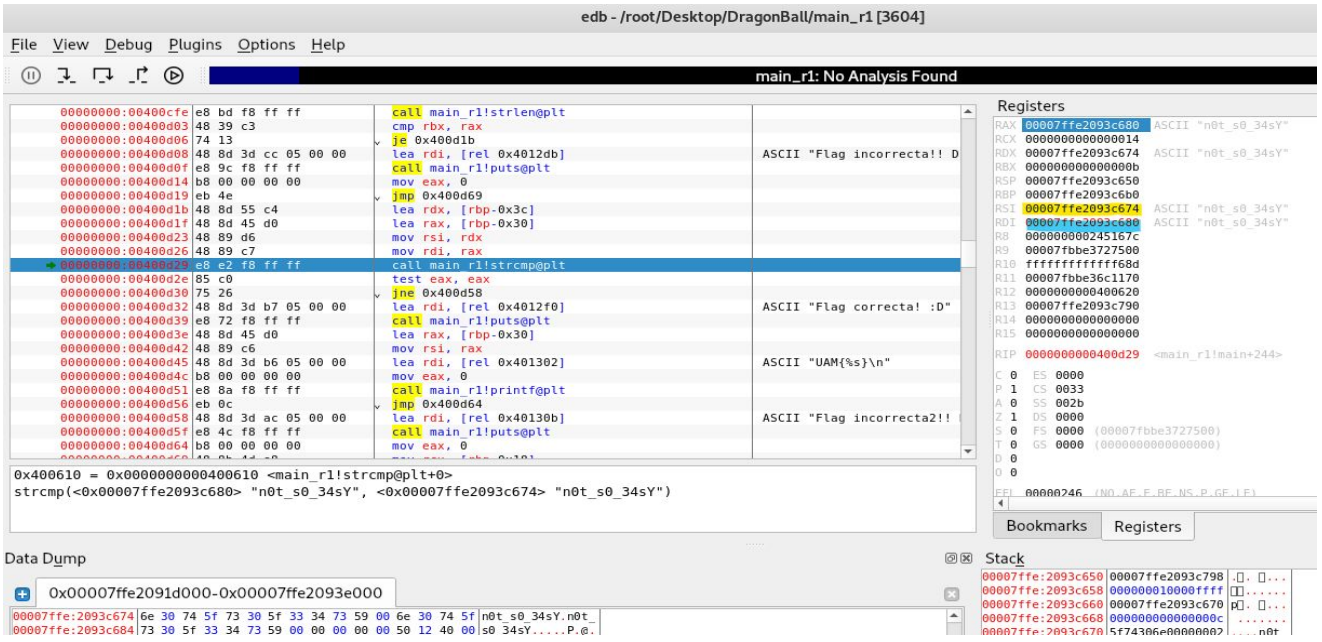
La parte interesante de esta función, es la utilización de xor 69h para toda la cadena (xor ecx, 69h).

[07,59,1d,36,1a,59,36,5a,5d,1a,30] ⊕ 69

[6e,30,74,5f,73,30,5f,33,34,73,59]

n0t_s0_34sY (11 caracteres 0b)

Procedemos a realizar la depuración con EDB.
Tras unos cuantos “F8”, llegamos a la función main 00400c35, seguimos, introducimos cadena “n0t_s0_34sy”, seguimos hasta llegar a la comparación de cadenas.



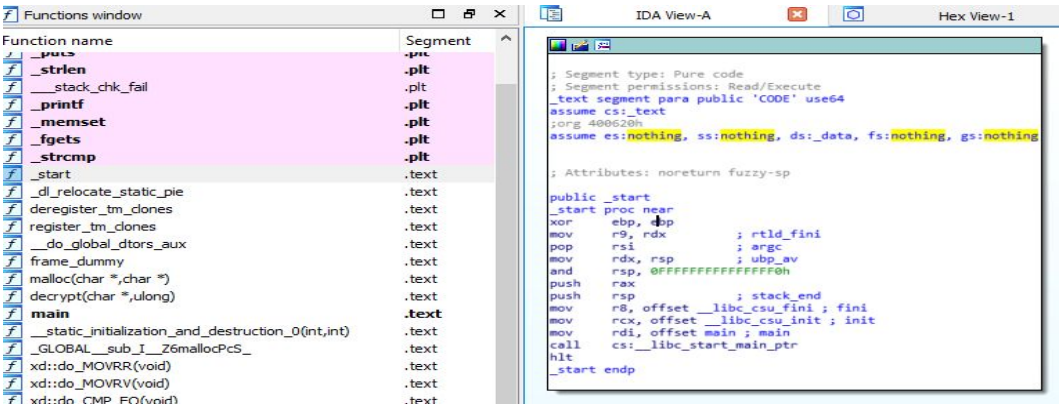
En la imagen tenemos resaltado en amarillo la cadena s2 y en azul la s1. en el momento de la comparacion (strcmp). El programa finaliza correctamente mostrando “Flag correcta! :D” UAM{n0t_s0_34sy}

Demasiado fácil….. (sobra decir que no es la flag)

Tendremos que seguir buscando. Volvemos a Ida

Si nos fijamos, disponemos muchas funciones que a priori no se utilizan xd:do_MOVRR, do_JMP……). Tendremos que analizar desde el principio, a ver que se nos escapa…

Empezamos en el Entry Point del programa (función **_start**) 0x400620h

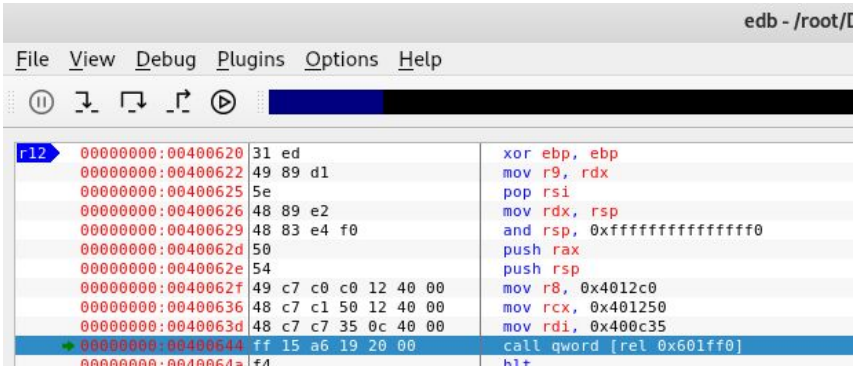


Utiliza una llamada a lib_c_start_main, pasándole 3 parámetros (main, csu_init, csu_finit). La siguiente página, nos explica detalladamente su funcionamiento.
http://dbp-consulting.com/tutorials/debugging/linuxProgramStartup.html

Tenemos que csu_init es un puntero a la función que realiza la inicialización inicial (constructor) y csu_fini otro al que se encarga de la inicialización final (destructor).

Lo importante es que la llamada al constructor se realiza antes de “llamar” main.

Esta vez, en EDB, nos quedamos en 400620,



En rcx tenemos el constructor (0x401250)

```
.text:0000000000401250 ; void _libc_csu_init(void)
.text:0000000000401250                 public __libc_csu_init
.text:0000000000401250 __libc_csu_init proc near               ; DATA XREF: _start+16↑o
.text:0000000000401250                 push    r15
.text:0000000000401252                 push    r14
.text:0000000000401254                 mov     r15, rdx
.text:0000000000401257                 push    r13
.text:0000000000401259                 push    r12
.text:000000000040125B                 lea     r12, __frame_dummy_init_array_entry
.text:0000000000401262                 push    rbp
.text:0000000000401263                 lea     rbp, __do_global_dtors_aux_fini_array_entry
.text:000000000040126A                 push    rbx
.text:000000000040126B                 mov     r13d, edi
.text:000000000040126E                 mov     r14, rsi
.text:0000000000401271                 sub     rbp, r12
.text:0000000000401274                 sub     rsp, 8
.text:0000000000401278                 sar     rbp, 3
.text:000000000040127C                 call    _init_proc
.text:0000000000401281                 test    rbp, rbp
.text:0000000000401284                 jz      short loc_4012A6
.text:0000000000401286                 xor     ebx, ebx
.text:0000000000401288                 nop     dword ptr [rax+rax+00000000h]
.text:0000000000401290
.text:0000000000401290 loc_401290:                             ; CODE XREF: __libc_csu_init+54↓j
.text:0000000000401290                 mov     rdx, r15
```

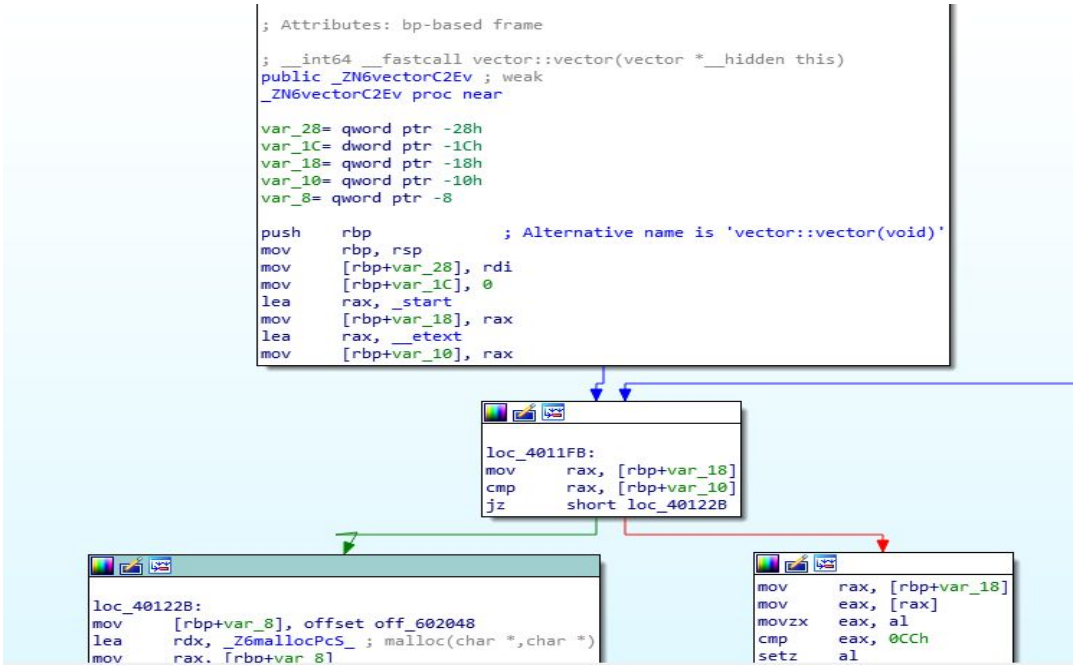
```
.init_array:0000000000601E08 __frame_dummy_init_array_entry dq offset frame_dummy
.init_array:0000000000601E08                                         ; DATA XREF: LOAD:00000000004000F8↑o
.init_array:0000000000601E08                                         ; LOAD:0000000000400210↑o ...
.init_array:0000000000601E08                                         ; Alternative name is '__init_array_start'
.init_array:0000000000601E10                 dq offset _GLOBAL__sub_I__Z6mallocPcS_
.init_array:0000000000601E10 _init_array     ends
```

Ya tenemos nuestra función inicial. **_GLOBAL__sub_I__Z6mallocPcS_**

La secuencia de llamadas sería:

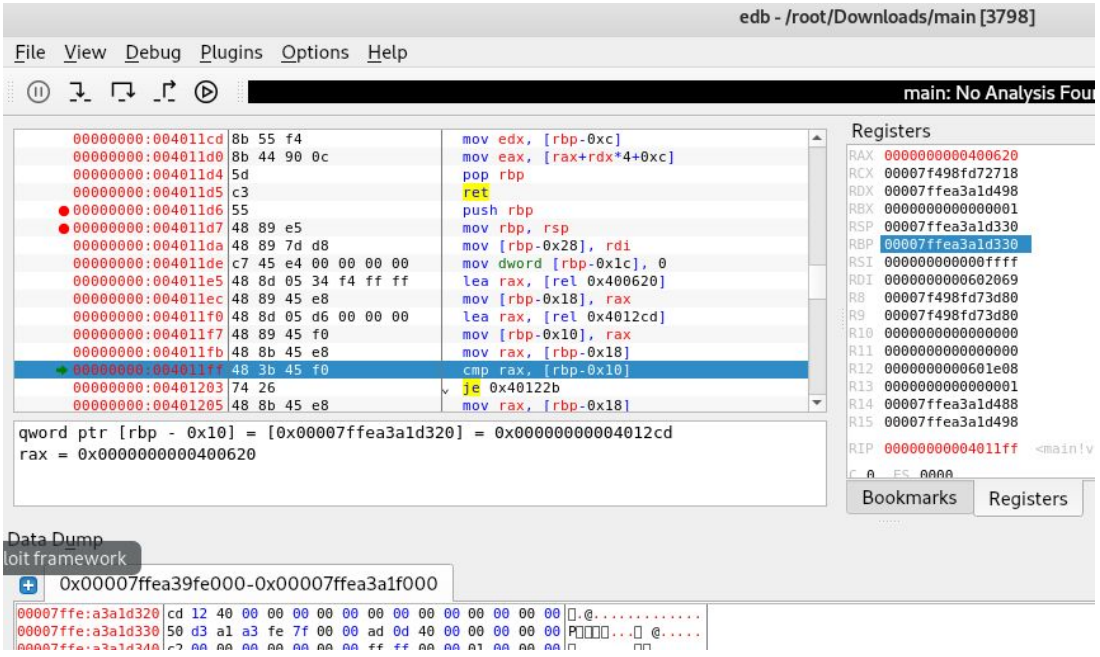
_GLOBAL__sub_I__Z6mallocPcS_ -> _Z41__static_initialization_and_destruction_0ii -> **_ZN6vectorC2Ev**.

_ZN6vectorC2Ev.



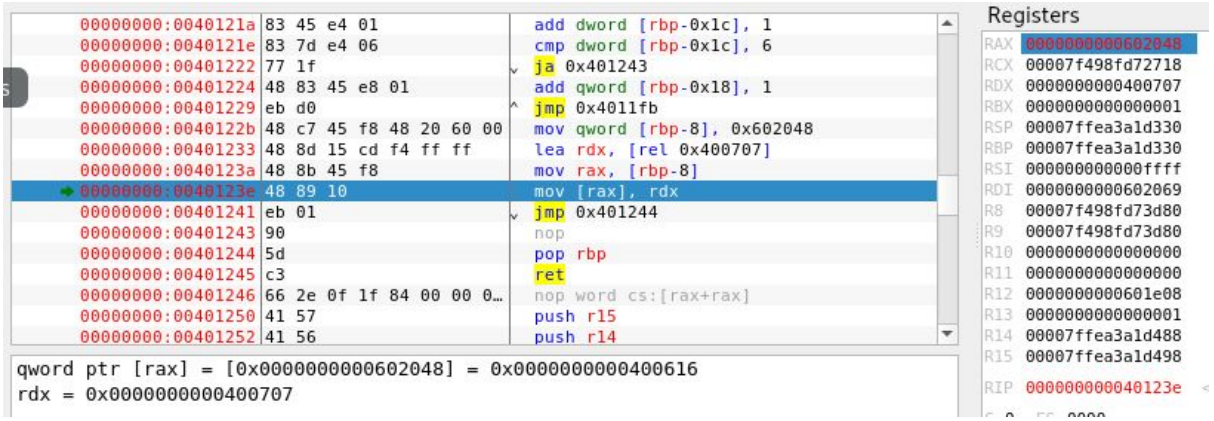
Interesante, aquí tenemos una llamada a **Z6mallocPcS**, malloc (char *,char *), esta función no es la “standard”, sino la definida en el programa, aunque está condicionada, veamos el flujo en el depurador.

Recargamos el ejecutable en EDB y ponemos punto de interrupción en _ZN6vectorC2Ev **0x4011D6**



[rbp-0x18] = rax = 400620 y [rbp-0x10] =4012cd. Luego no pasará por malloc… Modificamos rax y lo igualamos a 4012cd, para forzar el salto.

Continuamos:

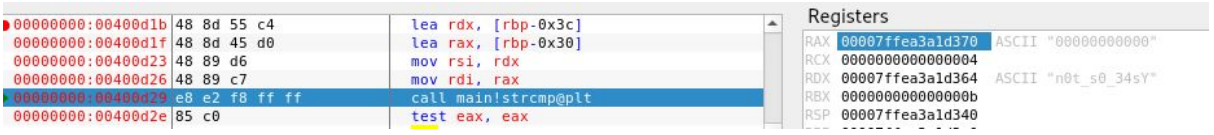


Aquí tenemos una parte de lo más interesante, lo que realiza es cargar en rdx el puntero a la función malloc (400707) y poner ese valor en 0x602048, y ¿esta dirección ?, la miramos en ida

.got.plt:0000000000602048 off_602048 dq offset strcmp ; DATA XREF: _strcmp↑r

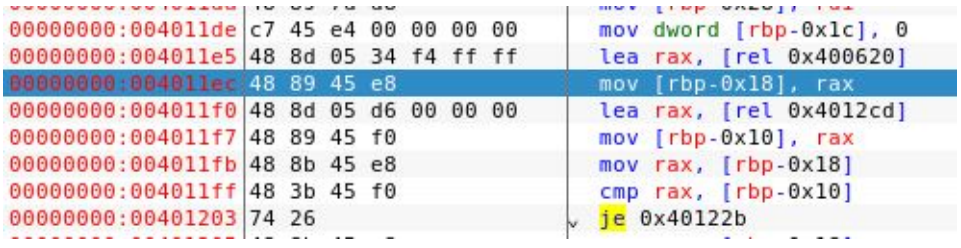
Modifica el puntero a la función strcmp y lo sustituye por malloc, lo que quiere decir, que en nuestra función main, cuando se llame a strcmp, llamará a malloc.

Para verificarlo, seguimos depuración y ponemos punto de interrupción en 400d1b, ponemos como cadena “00000000000”.



F7 -> F8 y BINGO!!, estamos en malloc 0x400707.

Llegado a este punto, para facilitar la depuración, y no tener que modificar rax, en _ZN6vectorC2Ev, a mano para que realice el salto de cambiar el puntero a malloc, parcheamos el ejecutable.



La idea es cambiar la instrucción mov rax,[rbp-0x18] por mov rax,[rbp-0x10] con lo que siempre serán iguales en el cmp. Para esto con un editor hexadecimal buscamos la cadena

“48 89 45 f0 48 8b 45 **e8** 48 3b 45 f0” y la sustituimos por “48 89 45 f0 48 8b 45 f0 48 3b 45 f0”. Realmente es cambiar el valor **e8** [rbp-0x18] por **f0** [rbp-0x10].

```
00000000:004011e5 48 8d 05 34 f4 ff ff    lea rax, [rel 0x400620]
00000000:004011ec 48 89 45 e8             mov [rbp-0x18], rax
00000000:004011f0 48 8d 05 d6 00 00 00    lea rax, [rel 0x4012cd]
00000000:004011f7 48 89 45 f0             mov [rbp-0x10], rax
00000000:004011fb 48 8b 45 f0             mov rax, [rbp-0x10]
00000000:004011ff 48 3b 45 f0             cmp rax, [rbp-0x10]
00000000:00401203 74 26                   je 0x40122b
```

Volvemos a la función malloc.

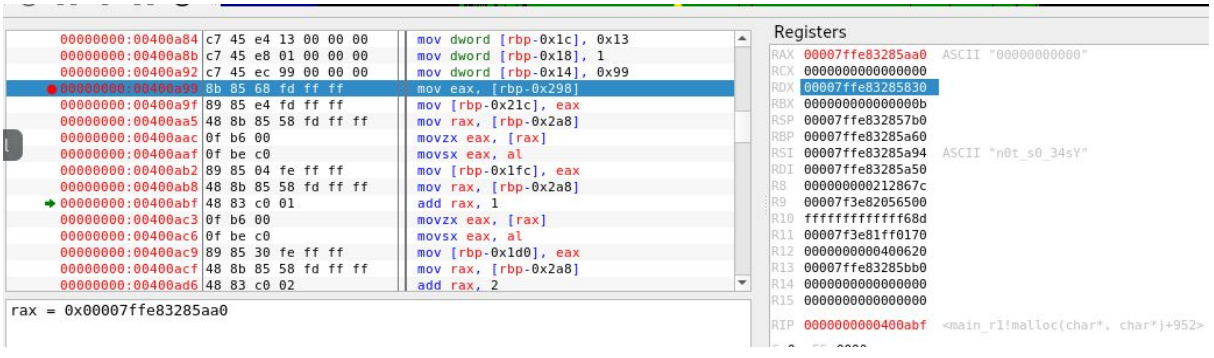
Tras analizarla, comprobamos que realiza una primera inicialización de valores, y llamadas a ciertas funciones xd:xd, xd:run.. xd::Movrr, etc.
Parece que nos encontramos en una especie de VM, que ejecuta una serie de operaciones.

En **_ZN2xd3RunEv** ; __int64 __fastcall **xd::Run**(xd *__hidden this) podemos ver las diferentes llamadas condicionales a funciones, en base a un valor.(opcode)

Resumiendo tenemos:
99h => RET (END)
55h => do_MOVRR
33h => do_CMP_EQ
44h => do_JMP_NEQ
77h => do_JMP
88h => do_XOR
69h => do_MOVRV

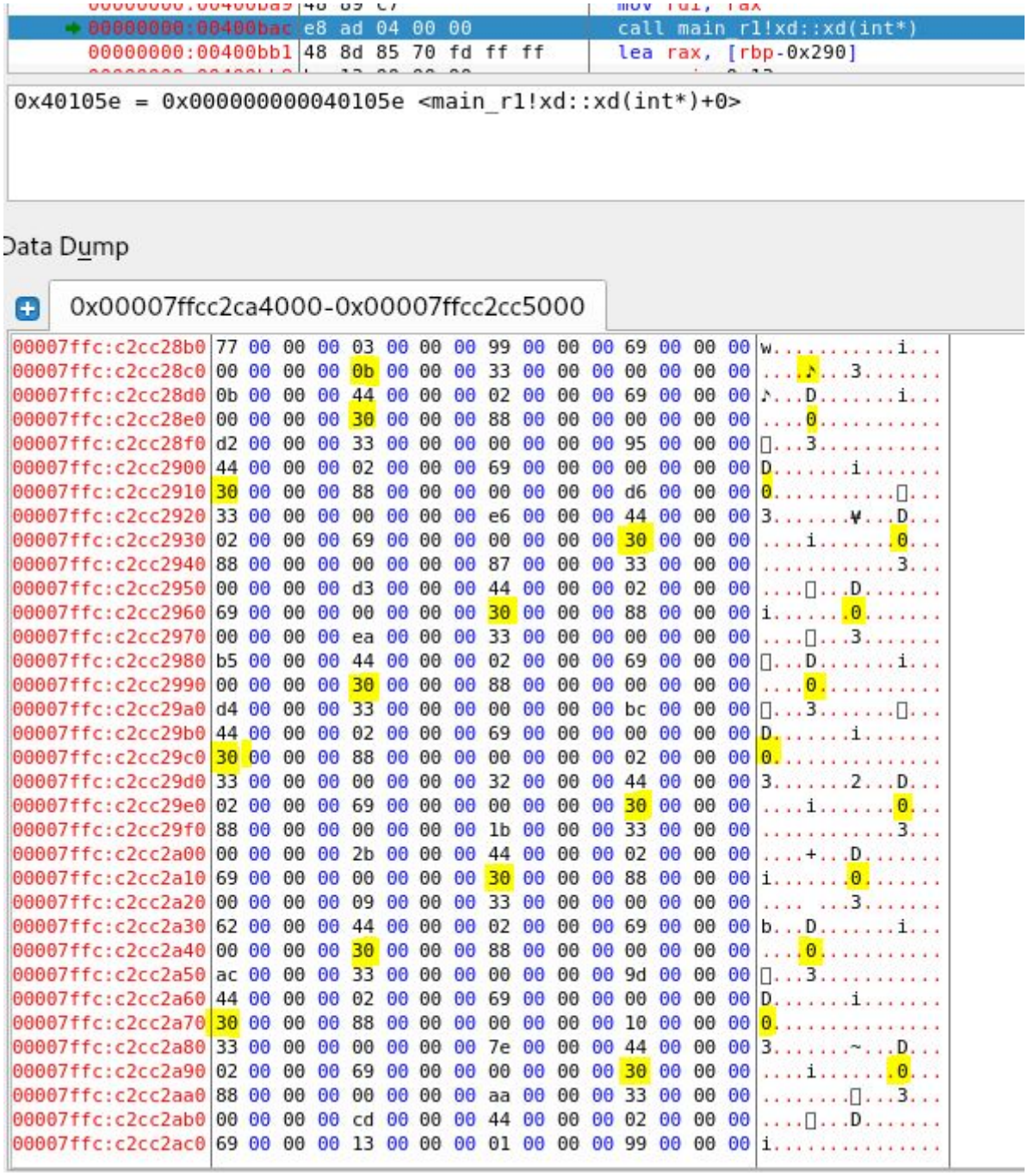
Ahora debemos analizar la inicialización de la VM, malloc, que es donde se establece el funcionamiento.

EDB, punto de interrupción 0x400707 y luego ejecutamos hasta 400a99.



Si seguimos la ejecución, vemos cómo se inicializa un vector con las instrucciones(opcodes) y después se utiliza la cadena “00000000000” y su tamaño, para insertarlas en diferentes zonas del vector, separando la cadena en caracteres individuales.

Aquí tenemos el volcado en memoria de las instrucciones de la VM, en amarillo está resaltado la parte de la cadena introducida por teclado (“00000000000” tamaño “0b”).



En psudocódigo sería algo así:

01 jmp 3 (77 00 00 00 03 00 00 00)
02 ret (99 00 00 00)
03 movrv 0b (69 00 00 00 00 00 00 00 0b 00 00 00)
04 cmp_eq 0b (33 00 00 00 00 00 00 00 0b 00 00 00)
05 jmp_ne 2 (44 00 00 00 02 00 00 00)

06 movrv 30 (69 00 00 00 00 00 00 00 30 00 00 00)
07 xor d2 (88 00 00 00 00 00 00 00 d2 00 00 00)
08 cmp_eq 95 (33 00 00 00 00 00 00 00 95 00 00 00)
09 jmp_ne 2 (44 00 00 00 02 00 00 00)

10 movrv 30 (69 00 00 00 00 00 00 00 30 00 00 00)
11 xor d6 (88 00 00 00 00 00 00 00 d6 00 00 00)
12 cmp_eq e6 (33 00 00 00 00 00 00 00 e6 00 00 00)
13 jmp_ne 2 (44 00 00 00 02 00 00 00)

….

Ya llegados a este punto, vemos que se repite la estructura, la idea es que comprobar si el tamaño de la cadena es 0b, si lo es, sigue, si no ret. Luego va cargando los dígitos de la cadena introducida, le realiza un xor y compara el valor, si correcto sigue, si no, pues termina.

Para extraer la flag, buscamos los xor y el valor comparado, operamos y obtenemos el valor original:

d2 ⊕ 95 => 47 G
d6 ⊕ e6 => 30 0
87 ⊕ d3 => 54 T
ea ⊕ b5 => 5F _
d4 ⊕ bc => 68 h
02 ⊕ 32 => 30 0
1b ⊕ 2b => 30 0
09 ⊕ 62 => 6b k
ac ⊕ 9d => 31 1
10 ⊕ 7e => 6e n
aa ⊕ cd => 67 g

Luego tenemos la flag => G0T_h00k1ng

Probamos en el ejecutable parcheado:

./main_r1
Flag: G0T_h00k1ng
Flag correcta! :D
UAM{G0T_h00k1ng}

UAM{7b02cd3d2d3cea80359cf600799413d3}